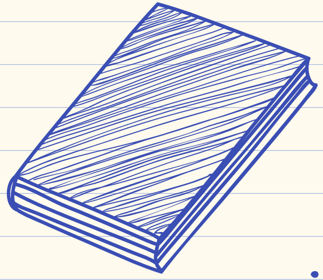
Quantum Computing
IBM CHALLENGE Lab 4

By Michael and Rebecca



Imports & Installations Needed

```
### Install Qiskit and relevant packages, if needed
### IMPORTANT: Make sure you are on 3.10 > python < 3.12
%pip install qiskit[visualization]==1.0.2
%pip install qiskit-ibm-runtime
%pip install qiskit-aer
%pip install graphviz
%pip install qiskit-serverless -U
%pip install qiskit-transpiler-service -U
%pip install git+https://github.com/qiskit-community/Quant
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import minimize

from qiskit import QuantumCircuit
from qiskit.quantum_info import SparsePauliOp
from qiskit.circuit.library import RealAmplitudes
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
from qiskit.transpiler import InstructionProperties
from qiskit.visualization import plot_distribution
from qiskit.providers.fake_provider import GenericBackendV2
from qiskit.primitives import StatevectorEstimator

from qiskit_aer import AerSimulator
from qiskit_ibm_runtime import (
    QiskitRuntimeService,
    EstimatorV2 as Estimator,
    SamplerV2 as Sampler,
    EstimatorOptions
)
```

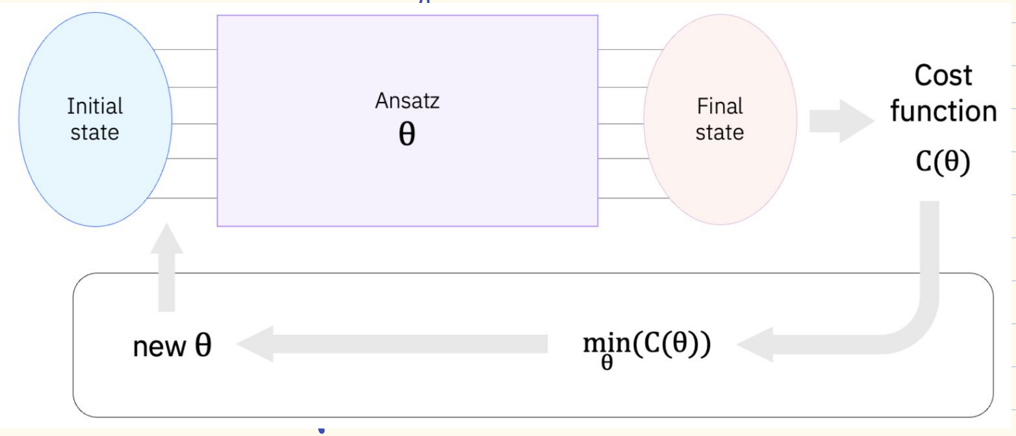
These libraries are essential for creating and running quantum circuits, handling data, visualizing results, and optimizing quantum algorithms.

What is a VQC - Variational Quantum Classifier

- They can solve certain types of classification problems.
- This architecture is based on an ansatz in the form of a parametrized quantum circuit applied onto an initial state.
- The output is measured in the form of a cost function.
- This cost function is classically optimized over the circuit's parameters.
- The optimization continues until we converge to a minimum.

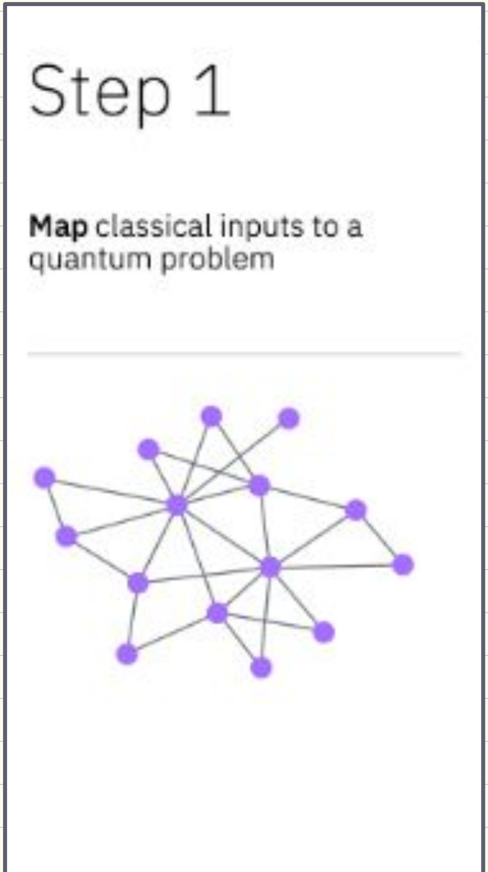
Algorithm that is able to find a good solution in a very large search space.

Quantum supremacy



Making a successful VQC

Take regular data and form it into something the quantum computer can understand (Map it)



Making a successful VQC

Adjust the quantum circuit's settings to make sure it runs efficiently. This step is like fine-tuning a machine to get the best performance.

Step 2

Optimize problem for quantum execution.

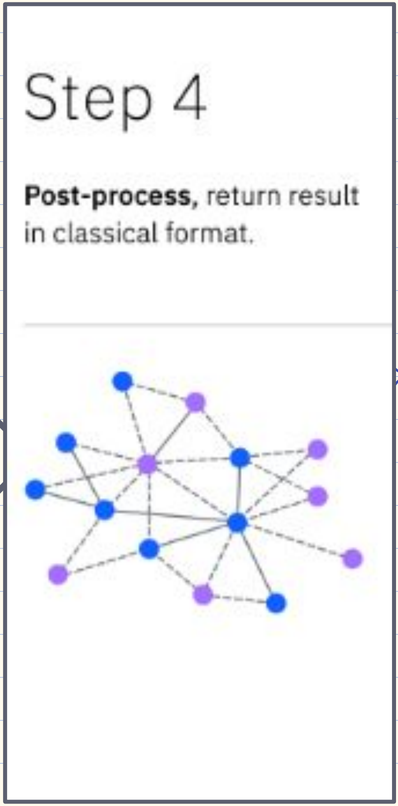
```
PassManager([UnitarySynthesis(),
             BasisTranslator(),
             EnlargeWithAncilla(),
             AISwap(),
             Collect1qRuns(),
             Optimize1qGates(),
             Collect2qBlocks(),
             ConsolidateBlocks()])
```

Making a successful VQC

Run the quantum circuit on a quantum computer, which processes the data and gives us results. (virtualize and use quantum hardware)

what is a VQC

We convert the quantum results back into a form that we can easily understand and use. (In our case



VQC's can help with tasks like recognizing images, understanding natural language, and predicting trends in finance.

Can be applied to any type of problem.

Checking the Dataset

```python
# Define num_qubits, the number of qubits, for the rest of the Lab
num_qubits = 5

# Load the dictionary
birds_dataset = pd.read_csv('birds_dataset.csv')

# Check if the dataset is loaded correctly - coefficients should be complex
for i in range(2**num_qubits):
    key = 'c%.0f' %i
    birds_dataset[key] = birds_dataset[key].astype(

# Print the dataset
birds_dataset
```

Code given to use with example of dataset shown.

1. Define the number of qubits
2. Load the dataset
3. Check to ensure that the coefficients are complex numbers, which are necessary for quantum computations.

	names	c0	c1	c2	c3	c4	c5	c6	c7	c8	...	c22	c23	c24	c25	
0	Falcon	0.707107+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
1	Hummingbird	0.000000+0.000000j	0.707107+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
2	Eagle	0.000000+0.000000j	0.000000+0.000000j	0.707107+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
3	Osprey	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.707107+0.000000j	0.000000+0.000000j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
4	Heron	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.707107+0.000000j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
5	Peacock	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	1.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
6	Parrot	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.0+0.0j	1.0+0.0j	0.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
7	Swan	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.0+0.0j	0.0+0.0j	1.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
8	Toucan	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.0+0.0j	0.0+0.0j	0.0+0.0j	1.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(
9	Cardinal	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.000000+0.000000j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	...	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.0+0.0j	0.(

Mapping inputs

```
# List to store the coefficient lists for each label
coefficients_lists = []

# Iterate over each row in the DataFrame
for index, row in birds_dataset.iterrows():
    # Extract coefficients as a list
    coefficients = row[1:].tolist()  # Skip the label column
    # Append the list of coefficients to the main list
    coefficients_lists.append(coefficients)

coefficients_lists
```

Same way to do the problem

```
list_coefficients = birds_dataset.iloc[:, 1:].values.tolist()
num_birds = len(birds_dataset)
half_num_birds = num_birds // 2

list_labels = [1] * half_num_birds + [0] * (num_birds - half_num_birds)
```

1. Extracts all the state vector coefficients from the dataset (excluding the first column, which contains the bird names) and converts them into a list of lists. Each list represents the coefficients for one bird.

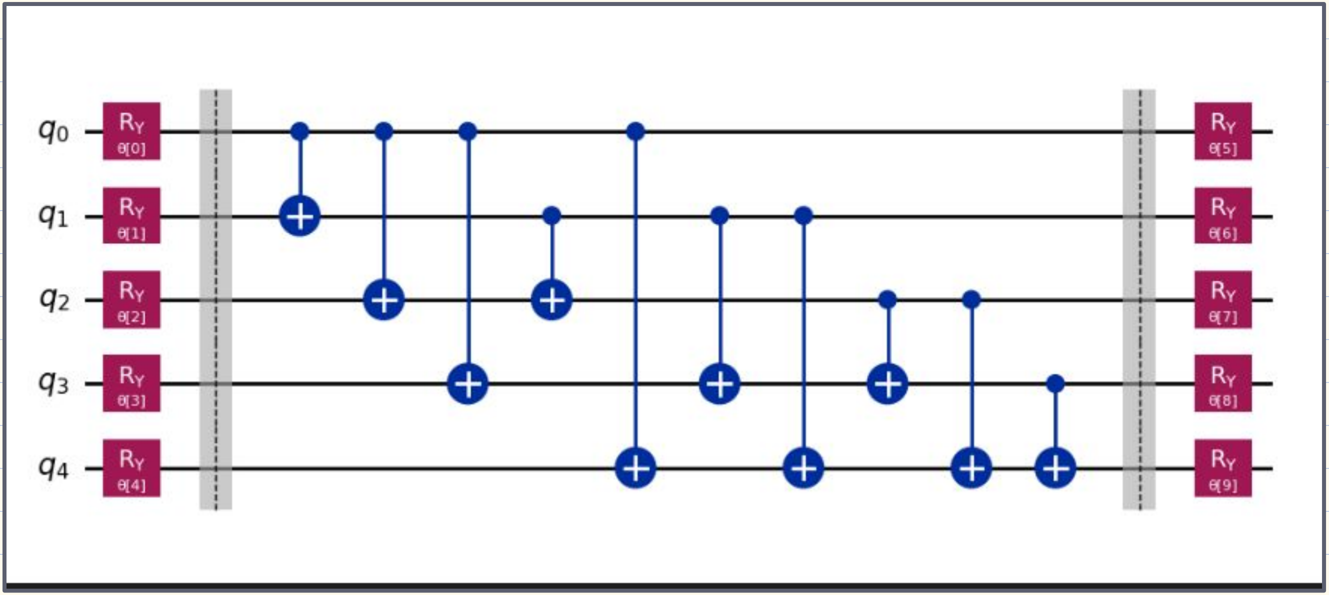
2. Count the total number of birds in the dataset

3. computes half the number of birds. This is useful for creating balanced labels for training the VQC.

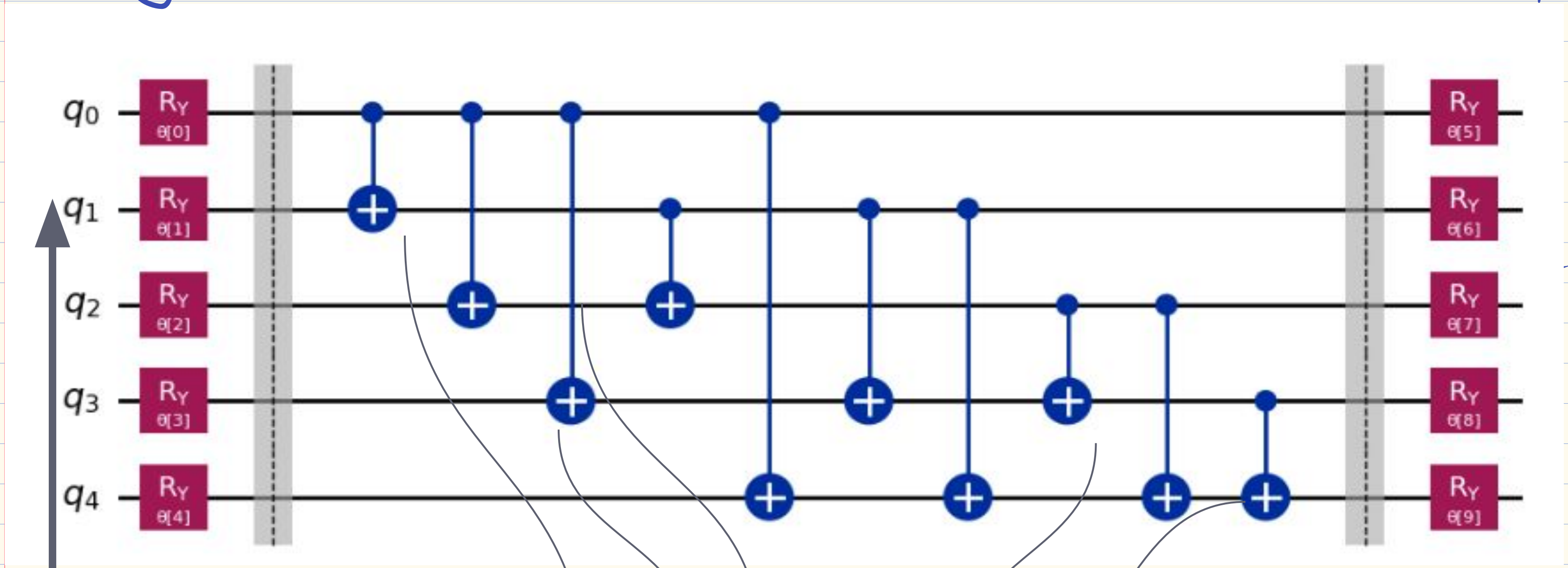
4. Create labels The first half of the birds are labeled 1, and the second half are labeled 0. This helps in training the VQC to distinguish between two categories (e.g., entangled vs. non-entangled states).

Building the Ansatz, Trade-offs

- Speed: By reducing the search space, and thus the number of gates and the depth of the ansatz, the algorithm can run faster.
- Accuracy: Reducing the search space could risk excluding the actual solution to the problem, leading to suboptimal solutions.
- Noise: Deeper circuits are affected by noise, so we need to experiment with our ansatz's connectivity, gates, and gate fidelity.



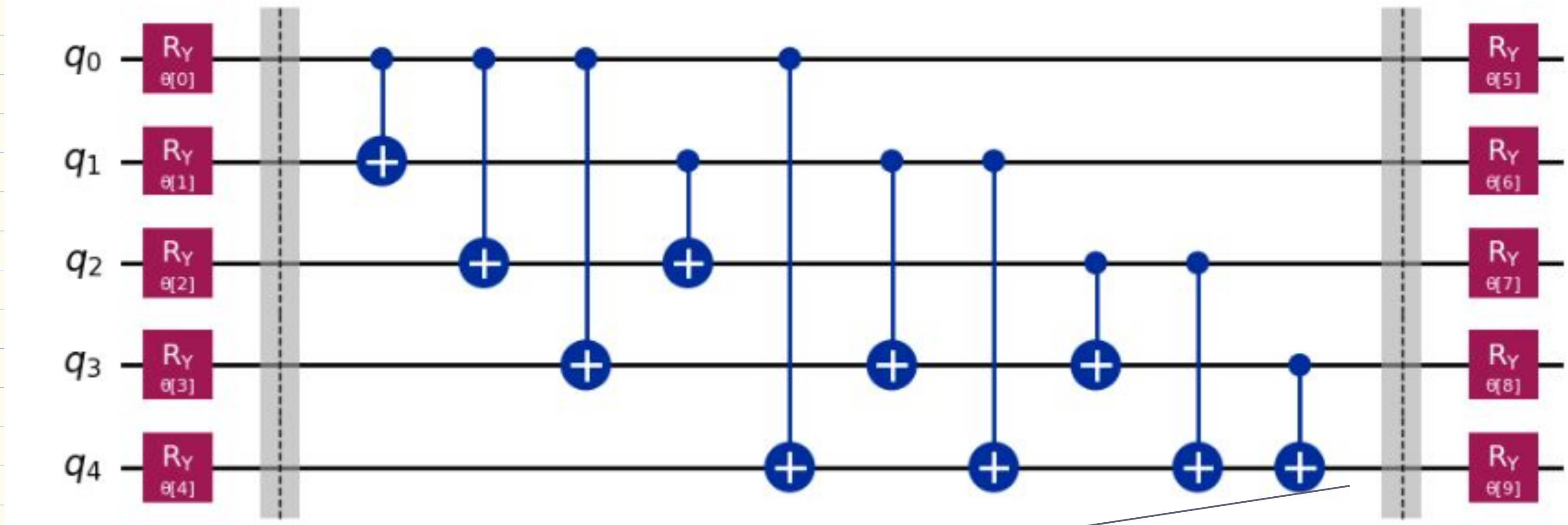
Building the Ansatz, Real Amplitudes (EXERCISE 2)



Q0-Q4 = 5 qubits

Full entanglement means that each qubit is entangled with every other qubit. This can be identified by examining the arrangement of the CNOT gates in the ansatz.

Building the Ansatz, Real Amplitudes



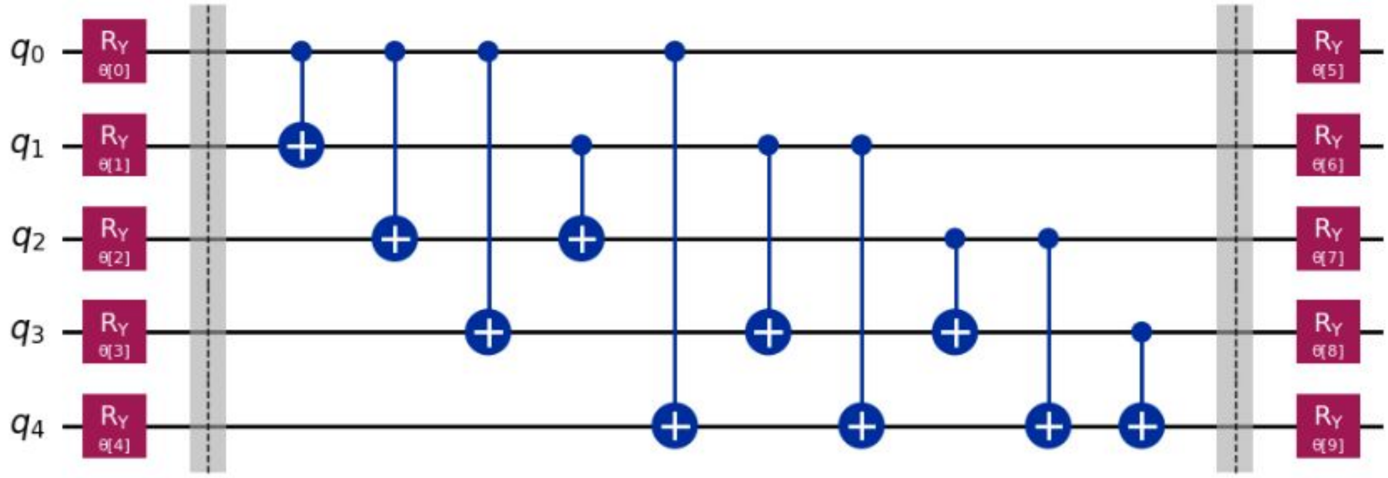
Repetition: refers to the number of times an operation is done on a qubit

```
num_qubits = 5  # Add your number of qubits here
reps = 1        # Add your number of repetitions here
entanglement = 'full'  # Add your entanglement strategy here
```

Building the Ansatz, writing the code.

```
# Create the ansatz using RealAmplitudes
ansatz = RealAmplitudes(num_qubits=num_qubits, reps=reps, entanglement=entanglement)
```

✓ 0.0s



Ansatz- refers to a parameterized quantum circuit designed to represent the quantum state of a system.

Optimize the Problem for Quantum Execution

Optimize the problem for quantum execution and execute it using Qiskit primitives.

In this exercise (3), you need to check which set of initial parameters yield the best convergence. For this, you need to define two functions that will be used throughout the Lab:

The cost function is defined in terms of the expectation value of an observable $\hat{O}$ on the outputs of the circuit for each of the birds in the dataset:

$$C(\theta) = \sum_{i \in \text{birds}} (\langle \psi_i(\theta) | \hat{O} | \psi_i(\theta) \rangle - L_i),$$

where $\psi_i(\theta)$ is the output state of the circuit for the bird i and L_i is the label for the same bird. The observable is $\hat{O} = ZZZZZ$ and θ is the vector of parameters for the ansatz.

If we successfully train the VQC, which means that we reach the optimal set of parameters θ^{opt} that minimizes the cost function, the VQC will output an expectation value of $\langle ZZZZZ \rangle = 1$ for IBM Quantum birds and $\langle ZZZZZ \rangle = 0$ for non-IBM Quantum birds.

Pass Manager and Cost Function

```python
obs = SparsePauliOp("ZZZZZ")
```

Define Observable with quantum circuit

```python
# Define the estimator and pass manager
estimator = StatevectorEstimator() #To train we use StatevectorEstimator to get the exact simulation
pm = generate_preset_pass_manager(backend=AerSimulator(), optimization_level=3, seed_transpiler=0)
```

State Vector Estimator- helps in computing the state vector of the quantum circuit.

```python
def cost_func(params, list_coefficients, list_labels, ansatz, obs, estimator, pm, callback_dict):

    """Return cost function for optimization

    Parameters:
        params (ndarray): Array of ansatz parameters
        list_coefficients (list): List of arrays of complex coefficients
        list_labels (list): List of labels
        ansatz (QuantumCircuit): Parameterized ansatz circuit
        obs (SparsePauliOp): Observable
        estimator (EstimatorV2): Statevector estimator primitive instanc
        pm (PassManager): Pass manager
        callback_dict (dict): Dictionary to store callback information

    Returns:
        float: Cost function estimate
    """
```

This function will be used to train the Variational Quantum Classifier (VQC) by finding the optimal parameters that minimize the cost function.

```python
cost = 0
for amplitudes,label in zip(list_coefficients, list_labels):
    qc = QuantumCircuit(num_qubits)
    # Amplitude embedding
    qc.initialize(amplitudes)
    # Compose initial state + ansatz
    classifier = qc.compose(ansatz)
    # Transpile classifier
    transpiled_classifier = pm.run(classifier)
    # Transpile observable
    transpiled_obs = obs.apply_layout(layout=transpiled_classifier.layout)
    # Run estimator
    pub = (transpiled_classifier, transpiled_obs, params)
    job = estimator.run([pub])
    # Get result
    result = job.result()[0].data.evs
    # Compute cost function (cumulative)
    cost += np.abs(result - label)

callback_dict["iters"] += 1
callback_dict["prev_vector"] = params
callback_dict["cost_history"].append(cost)

# Print the iterations to screen on a single line
print(
    "Iters. done: {} [Current cost: {}]".format(callback_dict["iters"], cost),
    end="\r",
    flush=True,
)
```

Defining the cost function cont.

Estimator uses the provided ansatz parameters to simulate the quantum circuit's state vector

state vector is then used to compute the expectation value of the observable (obs)

For each data point, using the complex coefficients and labels to evaluate the circuit's performance.

Classical Post Processing

```
# Intialize the lists to store the results from different ...
cost_history_list = []
res_list = []

# Retrieve the initial parameters
params_0_list = np.load("params_0_list.npy")

for it, params_0 in enumerate(params_0_list):

    print('Iteration number: ', it)

    # Initialize a callback dictionary
    callback_dict = {
        "prev_vector": None,
        "iters": 0,
        "cost_history": [],
    }

    # Minimize the cost function using scipy
    res = minimize(
        cost_func,
        params_0,
        args=(list_coefficients, list_labels, ansatz, obs, estimator, pm, callback_dict),
        method="cobyla", # Classical optimizer
        options={'maxiter': 200}) # Maximum number of iterations

    # Print the results after convergence
    print(res)

    # Save the results from different runs
```

This is the part in which we **TRAIN** the VQC.

To start the training, we need to define an initial set of parameters.

Printing the iteration number and initializing a callback dictionary for each iteration.

used to optimize the cost function.

Results from training

```
Iteration number:  0
 message: Optimization terminated successfully.82]]
 success: True
  status: 1
     fun: 4.057929433107382
       x: [ 3.739e+00 -1.799e-01  2.141e-02  3.719e+00  4.955e+00
            1.571e+00  2.475e-06  6.028e+00  2.534e+00  3.102e+00]
    nfev: 195
   maxcv: 0.0
Iteration number:  1
 message: Optimization terminated successfully.41]]
 success: True
  status: 1
     fun: 5.000060186657541
       x: [ 4.356e-03 -1.542e-04  3.449e-01 -3.696e-01  1.871e+00
            6.839e-01  1.566e+00  1.227e+00  1.105e+00 -9.954e-02]
    nfev: 117
   maxcv: 0.0
```

Convergence: All iterations terminated successfully, indicating that the optimizer found feasible solutions within the given constraints.

Performance: The function values ("fun") vary slightly across iterations, suggesting that the optimizer explored different regions of the parameter space.

Efficiency: The number of function evaluations ("nfev") indicates the computational effort required for each optimization run.

Choose Initial Parameters

A function `test_VQC`, which applies the circuit with optimal parameters to each of the birds in the data set and outputs the converged value of the cost function.

2. A function `compute performance` which outputs the total performance (P) for each set of optimal parameters, which is defined as

$$P = 100 - 100 \cdot \sum_{i \in \text{birds}} \frac{|\langle \psi_i(\theta_{opt})|\hat{O}|\psi_i(\theta_{opt})\rangle - L_i|}{2^5}.$$

Performance that is defined as:

Exercise 3: which set of initial parameters yield the best convergence?

```
est_VQC(list_coefficients, list_labels, ansatz, obs, opt_params, estimator, pm, num_qubits
""Return the converged value of the cost function for each bird in the dataset."""
esults_test = []
or amplitudes, label in zip(list_coefficients, list_labels):
    qc = QuantumCircuit(num_qubits)
    qc.initialize(amplitudes)
    classifier = qc.compose(ansatz)
    transpiled_classifier = pm.run(classifier)
    job = estimator.run(transpiled_classifier, [obs], opt_params)
    result = job.result()

    #print(result)

    evs = result.values
    results_test.append(evs[0])
print(results_test)
rint(opt_params)
eturn results_test
```

For each data point: New Quantum Circuit is created with the specified number of qubits.

The qubits are initialized with the given amplitudes. Classifying with The ansatz is composed with the initialized circuit to form the classifier.

The classifier circuit is transpiled using the pass manager (pm).

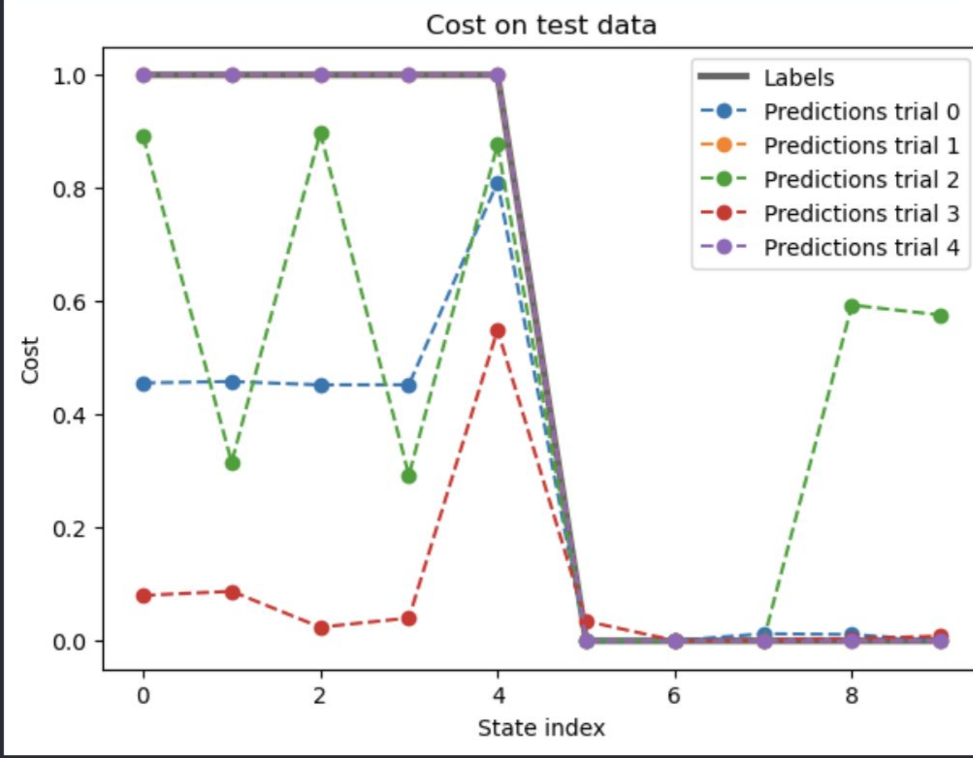
The estimator runs the transpiled classifier with the observable and optimized parameters to compute the expectation value.

Cost on Test Data

This function calculates the performance of the classifier by comparing the results obtained from the quantum classifier to the actual labels.

```
def compute_performance(result_list, list_labels):
    """Return the performance of the classifier."""
    total_cost = 0
    for result, label in zip(result_list, list_labels):
        total_cost += np.abs(result - label) / 25
    performance = 100 - 100 * total_cost / len(list_labels)
    return performance
0.0s
```

```
Performance for trial 0: 92.51500405825
Performance for trial 1: 99.99999696651076
Performance for trial 2: 90.95221049664029
Performance for trial 3: 86.66450481680403
Performance for trial 4: 99.99993709300665
```



Quantum Noise (simulation), Exercise 4

1. Decoherence: Qubits will lose their information of the quantum state over time, especially if they stay idle after they are initialized. For decoherence we normally don't use an error rate, but instead use T1 and T2 time, the amount of time it takes until a qubit loses its information.

2. Measurement Errors: Measuring qubits can cause errors, meaning that instead of a 0, a 1 is measured, and vice versa. This works similar to a classical channel.

3. Gate Errors: Gates are not perfect and have a small chance to introduce an error when applied. This is especially true for two-qubit gates, like the CX, the CZ, or the ECR gate, which normally have roughly a 10x higher error rate than single-qubit gates.

4. Crosstalk Errors: When applying a gate to a qubit, other qubits, especially neighboring ones, can also be influenced. This is even the case if these qubits lie idle. Fortunately, on the newest Heron devices, this is less of a problem, but it is still something to be aware of.

Simulating Quantum Noise.

```python
fake_backend = GenericBackendV2(
    num_qubits=5,
    basis_gates=["id", "rz", "sx", "x", "cx"]
    )
```

This code snippet sets up a fake backend for simulation purposes in Qiskit.

```python
def update_error_rate(backend, error_rates):

    """Updates the error rates of the backend

    Parameters:
        backend (BackendV2): Backend to update
        error_rates (dict): Dictionary of error rates

    Returns:
        None
    """
    default_duration=1e-8
    if "default_duration" in error_rates:
        default_duration = error_rates["default_duration"]

    # Update the 1-qubit gate properties
    for i in range(backend.num_qubits):
        qarg = (i,)
        if "rz_error" in error_rates:
            backend.target.update_instruction_properties('rz', qarg, InstructionProperties(error=error_rates["rz_error"], duration=default_duration))
        if "x_error" in error_rates:
            backend.target.update_instruction_properties('x', qarg, InstructionProperties(error=error_rates["x_error"], duration=default_duration))
        if "sx_error" in error_rates:
            backend.target.update_instruction_properties('sx', qarg, InstructionProperties(error=error_rates["sx_error"], duration=default_duration))
        if "measure_error" in error_rates:
            backend.target.update_instruction_properties('measure', qarg, InstructionProperties(error=error_rates["measure_error"], duration=default_durati

    # Update the 2-qubit gate properties (CX gate) for all edges in the chosen coupling map
    if "cx_error" in error_rates:
        for edge in backend.coupling_map:
            backend.target.update_instruction_properties('cx', tuple(edge), InstructionProperties(error=error_rates["cx_error"], duration=default_duration)
```

The update_error_rate function customizes the backend by setting error rates for both single and two-qubit gates, facilitating realistic simulations for quantum algorithm testing.

This function updates the error rates for various gates in the fake backend.

Testing the VQC on Fake Backend

whether we recognize the results from exercise 3.

- Test the VQC for different error rates for the `RZ` and `CX` gates. In each case, you will need to use the ```update_error_rate``` function.
- Use the optimal parameters from the best run of exercise 3.
- Compute the total performance `(P)` for each error rate using the function that you created previously and plot the final cost compared to the labels of each bird.

```
fig, ax = plt.subplots(1, 1, figsize=(7,5))
ax.set_title('Cost on test data')
ax.set_ylabel('Cost')
ax.set_xlabel('State index')
ax.plot(list_labels, 'k-', linewidth=3, alpha=0.6, label='Labels')

error_rate_list = [1e-1, 1e-2, 1e-3, 1e-4]

fake_backend = GenericBackendV2(
    num_qubits=5,
    basis_gates=["id", "rz", "sx", "x", "cx"]
)
```

Defining error rates and setting up fake backend

Backend is initialized with 5 qubits

Varying Error Rates

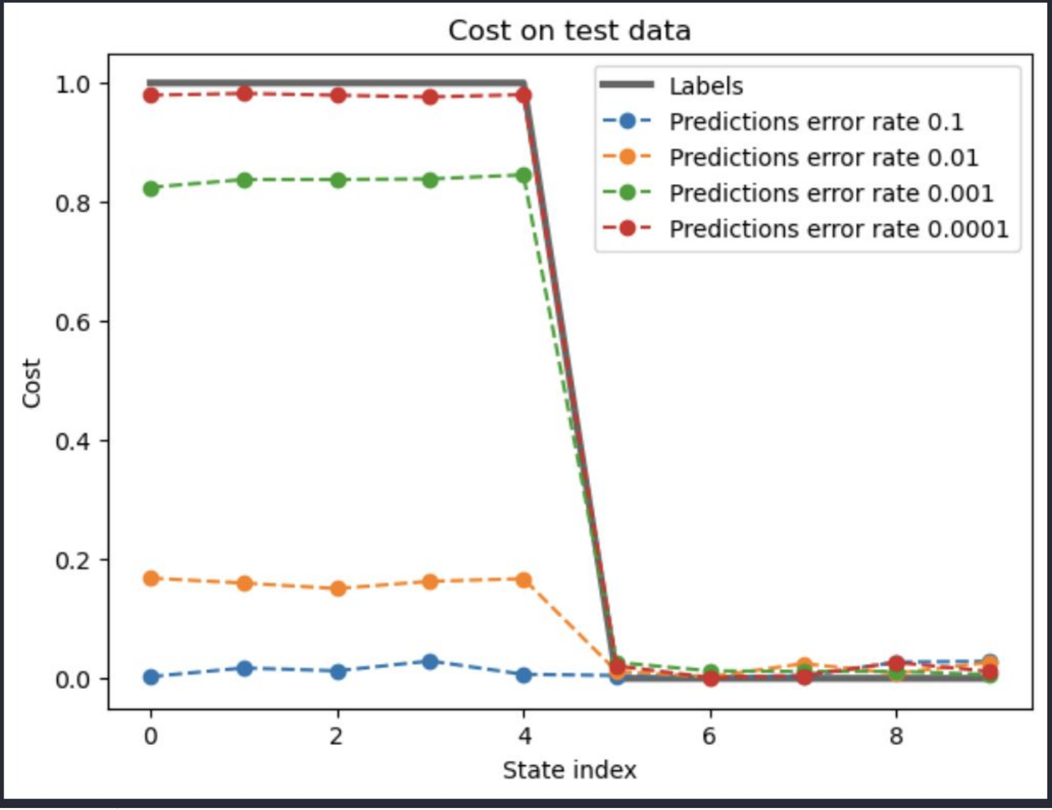
```
for error_rate_value in error_rate_list:
    update_error_rate(fake_backend, error_rates= {
    "default_duration": 1e-8,
    "rz_error": error_rate_value,
    "x_error": 1e-8,
    "sx_error": 1e-8,
    "measure_error": 1e-8,
    "cx_error": error_rate_value})


    estimator = Estimator()
    pm = generate_preset_pass_manager(optimization_level=3, backend=fake_backend)

    opt_params = res_list[4].x
    results_test = test_VQC(list_coefficients, list_labels, ansatz, obs, opt_params, estimator, pm,num_qubits)
    print(results_test)
    print(f"Performance for run {index}: {compute_performance(results_test, list_labels)}")
    ax.plot(results_test, 'o--', label='Predictions error rate '+str(error_rate_value))

ax.legend()
```

Instructions asked only for RZ AND CX

Everything else set with default value

generated graph

```
Performance for run 4: 84.3780517578125
Performance for run 4: 86.6729736328125
Performance for run 4: 97.23663330078125
Performance for run 4: 99.47662353515625

<matplotlib.legend.Legend at 0x7a02e4906dd0>
```



we are ready for Quantum Hardware

```
# Choose a real backend
service = QiskitRuntimeService()
backend = service.backend("ibm_rensselaer")

# Define a fake backend with the same properties as the real backend
fake_backend = AerSimulator.from_backend(backend)
```

Transpiling the circuit

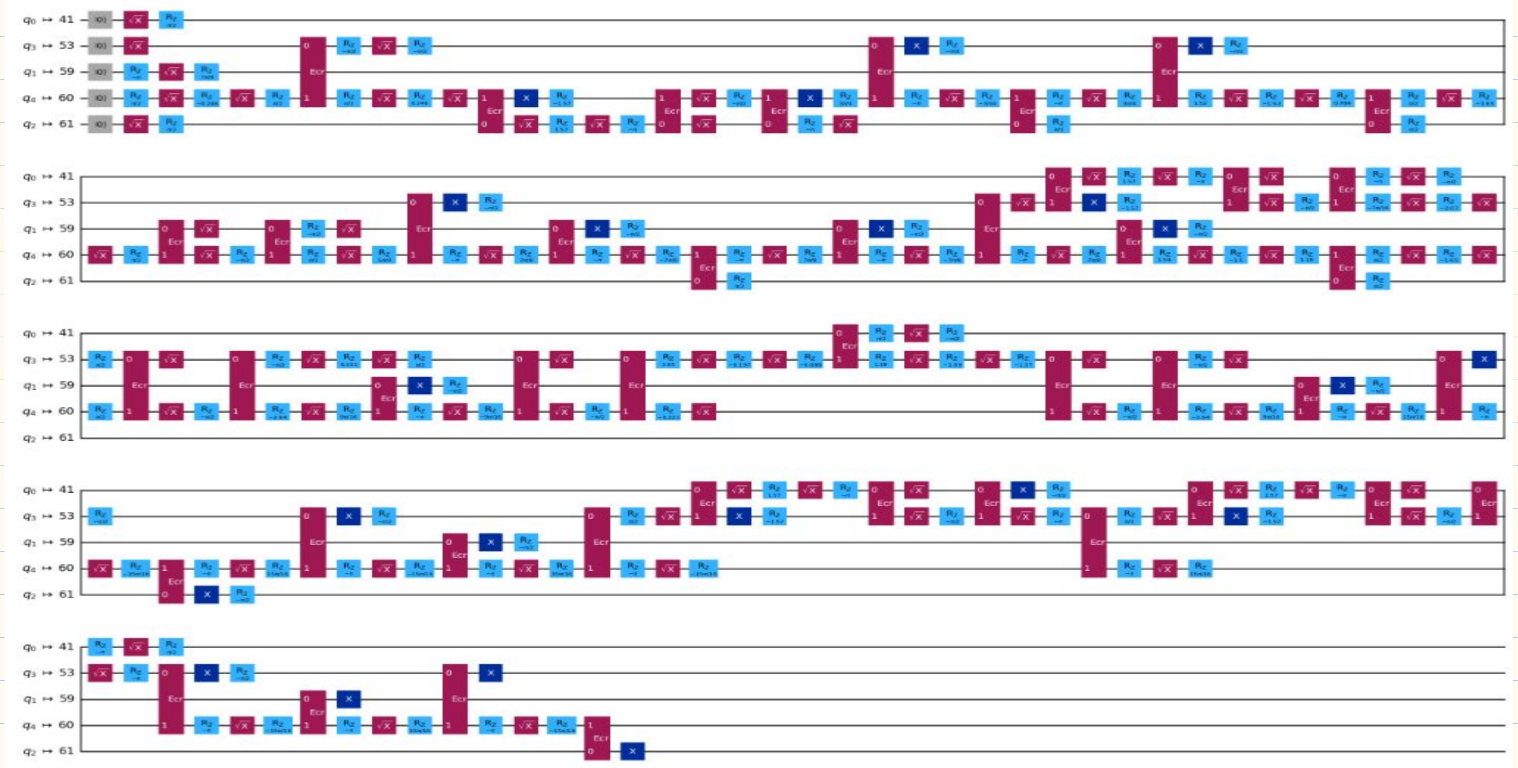
- Checking the two-qubit depth of circuit we get by using `qc.initialize` after transpilation.

```
index_bird = 0 #you can check different birds by changing the index
qc = QuantumCircuit(num_qubits)
qc.initialize(list_coefficients[index_bird])
pm = generate_preset_pass_manager(optimization_level=3, backend=fake_backend)
transpiled_qc = pm.run(qc)

print('Depth of two-qubit gates: ', transpiled_qc.depth(lambda x: len(x.qubits) == 2))
transpiled_qc.draw(output="mpl", idle_wires=False, fold=40)
```

✓ 2.5s

Our depth for the two qubits:
43

This circuit is too deep to be run on real quantum hardware!



Exercise 5:

Create a function to map the states more efficiently. For this we need to look closer at the data set and understand how it was built. whenever we want to run our code on the hardware, we always need to use the structure of our data and our problem wisely!

Create a function to map the states more efficiently.

```python
def amplitude_embedding(num_qubits, bird_index):
    """Create amplitude embedding circuit

    Parameters:
        num_qubits (int): Number of qubits for the ansatz
        bird_index (int): Data index of the bird

    Returns:
        qc (QuantumCircuit): Quantum circuit with amplitude embedding of the bird
    """
```

```python
def generate_GHZ(qc):
    qc.h(0)
    for i in range(num_qubits - 1):
        qc.cx(i, i + 1)

qc = QuantumCircuit(num_qubits)

if bird_index < 5:
    # IBM Quantum birds
    generate_GHZ(qc)
    binary_representation = f'{bird_index:05b}'
    for i, bit in enumerate(binary_representation):
        if bit == '1':
            qc.x(num_qubits - 1 - i)
else:
    # Non-IBM Quantum birds
    binary_representation = f'{bird_index:05b}'
    for i, bit in enumerate(binary_representation):
        if bit == '1':
            qc.x(num_qubits - 1 - i)

return qc
```

Helper function for amplitude embedding

This leads to a final GHZ state will be |0010> + |11101> for the bird number 2
|00011> + |11100> for bird 3
Non IBM Quantum birds correspond to the last 5 entries of the dict, with indices 5,6,7,8,9

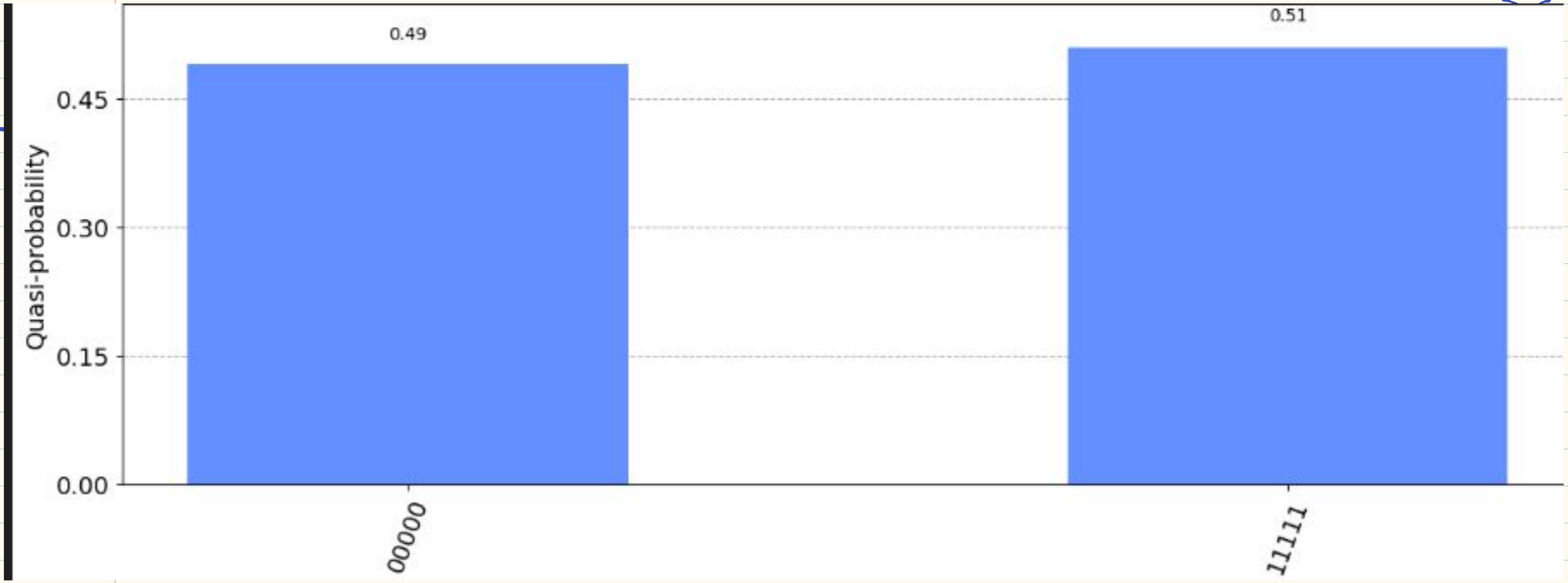
We start by generating a GHZ state starting using the function ```generate_GHZ```.

```
index_bird = 0 # You can check different birds by changing the index

# Build the amplitude embedding
qc = amplitude_embedding(num_qubits, index_bird)
qc.measure_all()

# Define the backend and the pass manager
aer_sim = AerSimulator()
pm = generate_preset_pass_manager(backend=aer_sim, optimization_level=3)
isa_circuit = pm.run(qc)

# Define the sampler with the number of shots
sampler = Sampler(backend=aer_sim)
result = sampler.run([isa_circuit]).result()
samp_dist = result[0].data.meas.get_counts()
plot_distribution(samp_dist, figsize=(15, 5))
```

✓ 0.1s

How to embed data into a quantum circuit, measure the qubits, and visualize the measurement outcomes.

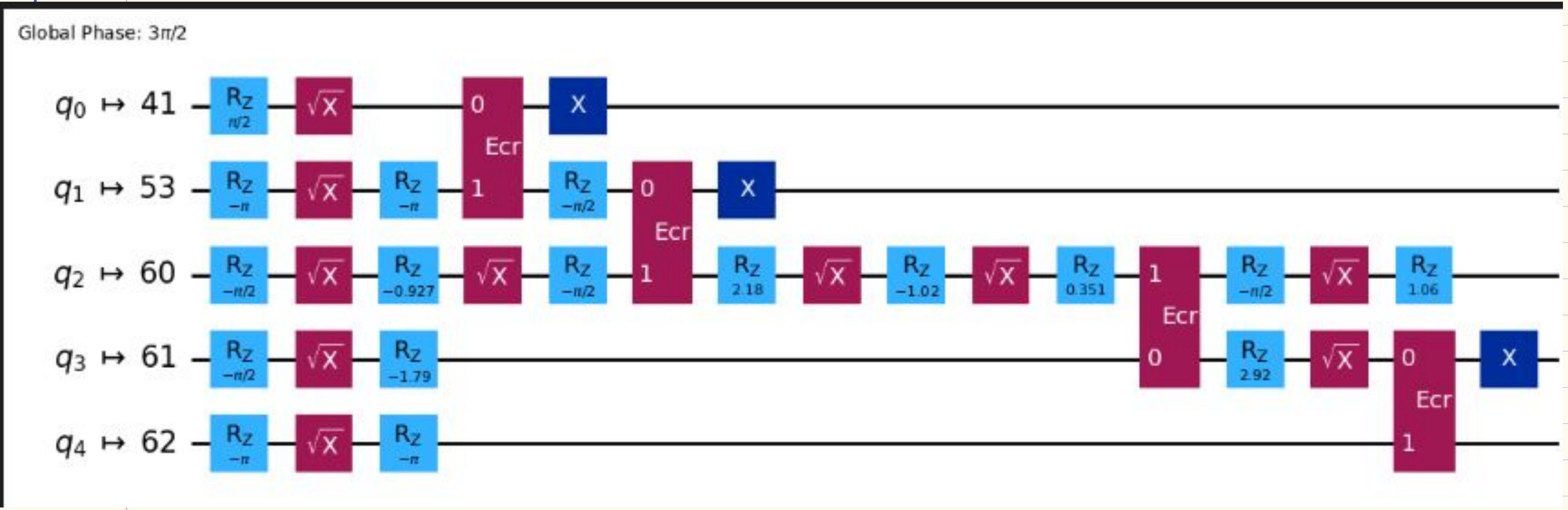
1. Selects the index of the bird to check
2. Builds the quantum circuit with amplitude embedding for the specified qubit index and adds measurement operations to all qubits.
3. Define the Aer simulator backend and pass manager for optimization.
4. Sets up the sampler with the Aer simulator backend, runs the circuit, and retrieves the measurement results.



The plot indicates that the state 11111 has a slightly higher probability (0.51) compared to the state 00000 (0.49).

The x-axis shows the measured quantum states (00000 and 11111), and the y-axis shows their corresponding probabilities.

Depth of two-qubit gates: 4



Optimization.

```
index_bird = 0 #You can check different birds by changing the index
qc = amplitude_embedding(num_qubits, index_bird)
pm = generate_preset_pass_manager(optimization_level=3, backend=fake_backend)
transpiled_qc = pm.run(qc)

print('Depth of two-qubit gates: ', transpiled_qc.depth(lambda x: len(x.qubits) == 2))
transpiled_qc.draw(output="mpl", fold=False, idle_wires=False)
```

✓ 0.4s

We have now optimized it so the depth of our qubits

Depth of two-qubit gates: 4

check the depth of the new amplitude embedding circuit

Depth of Transpiled version

Now, let's check the transpiled version of the ```RealAmplitudes``` ansatz using full connectivity.

```
old_ansatz = RealAmplitudes(num_qubits, reps=1, entanglement='full', insert_barriers=True)
pm = generate_preset_pass_manager(optimization_level=3, backend=fake_backend)
transpiled_ansatz = pm.run(old_ansatz)

print('Depth of two-qubit gates: ', transpiled_ansatz.depth(lambda x: len(x.qubits) == 2))
transpiled_ansatz.draw(output="mpl", idle_wires=False, fold=40)
```

Depth of two-qubit gates: 16

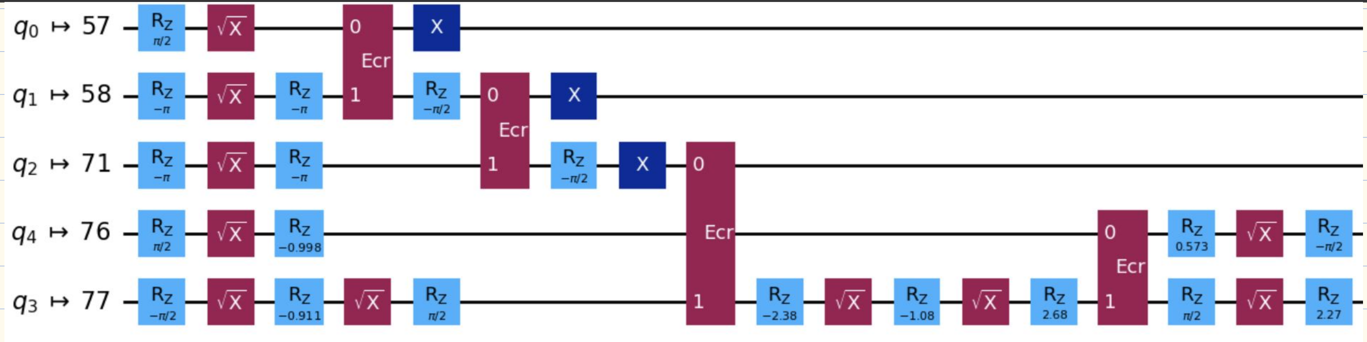
Depth of two-qubit gates: 16

Check the Reduction in Depth

Change the connectivity to a pairwise structure and check the depth of the circuit again.

```
ansatz = RealAmplitudes(num_qubits=num_qubits, reps=reps, entanglement=entanglement)
pm = generate_preset_pass_manager(optimization_level=3, backend=fake_backend)
transpiled_ansatz = pm.run(ansatz)

print('Depth of two-qubit gates: ', transpiled_ansatz.depth(lambda x: len(x.qubits) == 2))
transpiled_ansatz.draw(output="mpl", fold=False, idle_wires=False)
```

✓ 1.3s



Depth of two-qubit gates: 4

Compare the Depths

Old depth of two-qubit gates: 66
Current depth of two-qubit gates: 6

With the new ansatz we have reduced the depth by a factor of 10! This means that we are ready to test our VQC on quantum hardware.

```
old_mapping = QuantumCircuit(num_qubits)
old_mapping.initialize(list_coefficients[index_bird])
old_classifier = old_mapping.compose(old_ansat (variable) index_bird: int

new_mapping = amplitude_embedding(num_qubits, index_bird)
new_classifier = new_mapping.compose(ansatz)

pm = generate_preset_pass_manager(optimization_level=3, backend=fake_backend)
old_transpiled_classifier = pm.run(old_classifier)
new_transpiled_classifier = pm.run(new_classifier)

print('Old depth of two-qubit gates: ', old_transpiled_classifier.depth(lambda x: len(x.qubits) == 2))
print('Current depth of two-qubit gates: ', new_transpiled_classifier.depth(lambda x: len(x.qubits) == 2))
```

Exercise 6:

Create a new '''test_shallow_VQC''' function with the updated circuit.

- implement the new amplitude embedding step and the new ansatz. -
- we do not need the list of the coefficients anymore, since we are mapping each bird directly by its index.

Testing New Amplitude Embedding

```python
def test_shallow_VQC(list_labels, ansatz, obs, opt_params, estimator, pm):

    """Return the performance of the classifier

    Parameters:
        list_labels (list): List of labels
        ansatz (QuantumCircuit): Parameterized ansatz circuit
        obs (SparsePauliOp): Observable
        opt_params (ndarray): Array of optimized parameters
        estimator (EstimatorV2): Statevector estimator
        pm (PassManager): Pass manager for transpilation

    Returns:
        results_test (list): List of test results
```

For each label:

Amplitude Embedding: Embed amplitude and add measurement operations.

Classifier Composition: Combine embedding with ansatz.

Transpilation: Transpile using pass manager.

Estimation: Run estimator, retrieve results.

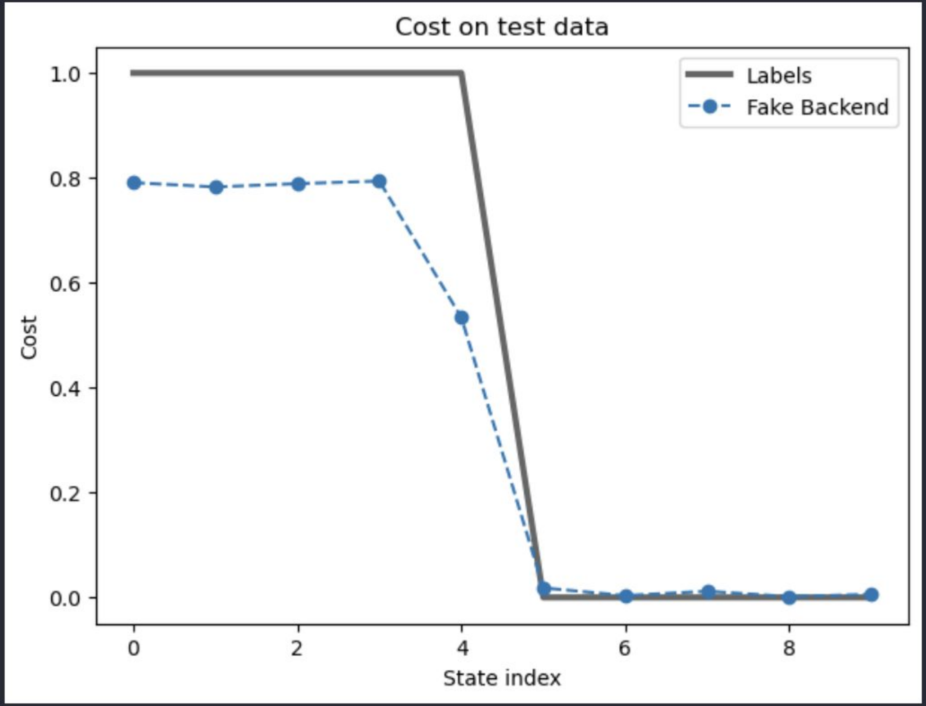
Results Storage: Append result to results_test.

```python
### Write your code below here ###
for index, label in enumerate(list_labels):
    # Create the amplitude embedding for the given index
    qc = amplitude_embedding(5, index)
    classifier = qc.compose(ansatz)
    transpiled_classifier = pm.run(classifier)
    job = estimator.run([transpiled_classifier], [obs], [opt_params])
    result = job.result()

    evs = result.values
    results_test.append(evs[0])


### Don't change any code past this line ###
return results_test
```

Results of shallow VQC

Performance: 95.78094482421875

<matplotlib.legend.Legend at 0x7a02d2183650>



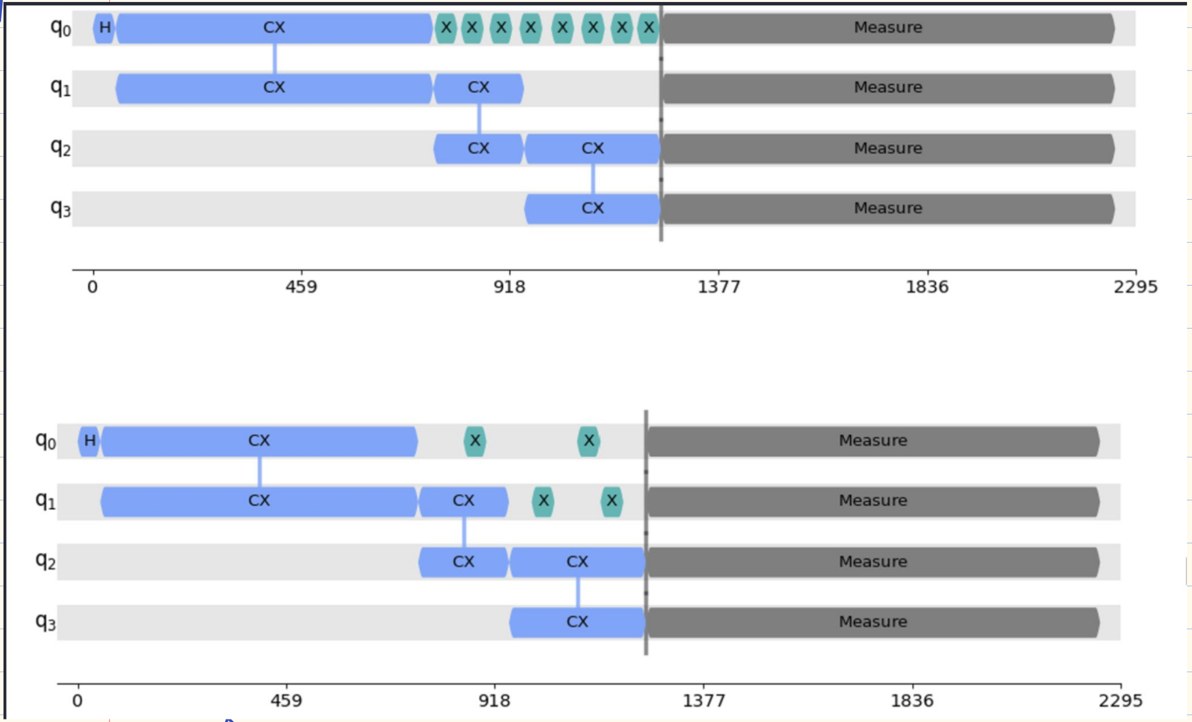
Exercise 7:

- Create a function to test the shallow VQC on the IBM Quantum backend.
- The function should be similar to ```test_shallow_VQC```, but now we need to create a list of pubs containing a pub for each bird ```pub = (transpiled_classifier, transpiled_obs, opt_params)```.
- Then, we call the Estimator primitive using the list of pubs and print the job ID so that you can retrieve it later.

Error Mitigation Techniques

Options	Sub-options	Sub-sub-options	Choices	Default
default_shots				`4096`
optimization_level			`0` / `1`	`1`
resilience_level			`0` / `1` / `2`	`1`
dynamical_decoupling	enable		`True` / `False`	`False`
	sequence_type		`'XX'` / `'XpXm'` / `'XY4'`	`'XX'`
	extra_slack_distribution		`'middle'` / `'edges'`	`'middle'`
	scheduling_method		`'asap'` / `'alap'`	`'alap'`
resilience	measure_mitigation		`True` / `False`	`True`
	measure_noise_learning	num_randomizations		`32`
		shots_per_randomization		`'auto'`
	zne_mitigation		`True` / `False`	`False`
	zne	noise_factors		`(1, 3, 5)`
		extrapolator	`'exponential'` / `'linear'` / `'double_exponential'` / `'polynomial_degree_(1 <= k <= 7)'`	(`'exponential'`, `'linear'`)
twirling	enable_gates		`True` / `False`	`False`
	enable_measure		`True` / `False`	`True`
	num_randomizations			`'auto'`
	shots_per_randomization			`'auto'`
	strategy		`'active'` / `'active-circuit'` / `'active-accum'` / `'all'`	`'active-accum'`

Dynamical Decoupling (DD)



Qubits can lose their information over time due to decoherence and can further be influenced by operations applied to other qubits via cross talk.

To eliminate these effects, we can use dynamic decoupling, which adds pulse sequences (known as dynamical decoupling sequences) to flip idle qubits around the Bloch sphere, canceling the effect of noise channels and thereby suppressing the decoherence effect.

In these 2 graphics below we can see X-gates being applied on qubits which are idle. Since we apply an even number of X-gates, the result is the identity and thus the effect of the X-gates cancel each other.

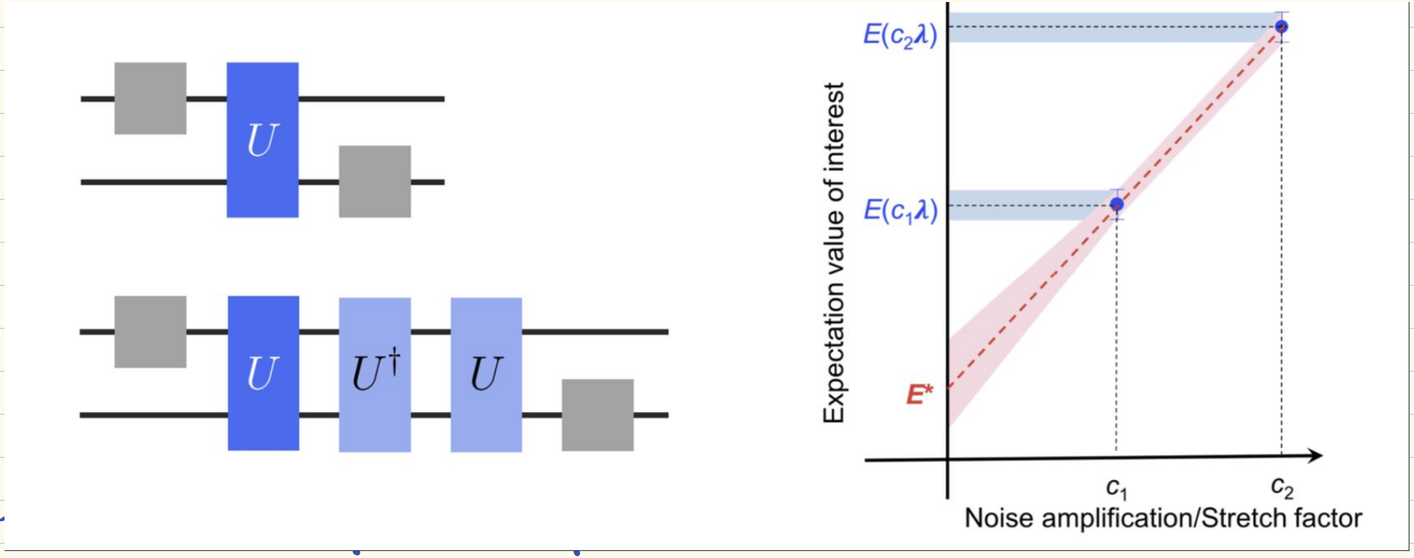
Twirled Readout Error extinction (TREX)

The TREX technique reduces the measurement error by diagonalizing the noise channel associated with measurement. The channel is obtained by randomly flipping qubits through X gates immediately before measurement. A rescaling term from the diagonal noise channel is learned by benchmarking random circuits initialized in the zero state. This allows the service to remove bias from expectation values that result from readout noise.

Zero Noise Extrapolation (ZNE)

Zero noise extrapolation is an error mitigation technique that can be used with the Estimator primitive. It has two distinct phases:

- In the first phase, the expectation value is calculated with different noise levels by amplifying the noise in the circuit.
- In the second step, the results are used to extrapolate what the expectation value would be without noise, so with zero noise.



Testing Error Mitigation Techniques

```python
service = QiskitRuntimeService()
backend = service.backend("ibm_rensselaer")
```

```python
def test_shallow_VQC_QPU(list_labels, anstaz, obs, opt_params, options, backend):

    """Return the performance of the classifier

    Parameters:
        list_labels (list): List of labels
        ansatz (QuantumCircuit): Parameterized ansatz circuit
        obs (SparsePauliOp): Observable
        opt_params (ndarray): Array of optimized parameters
        options (EstimatorOptions): Estimator options
        backend (service.backend): Backend to run the job

    Returns:
        job_id (str): Job ID
    """
```

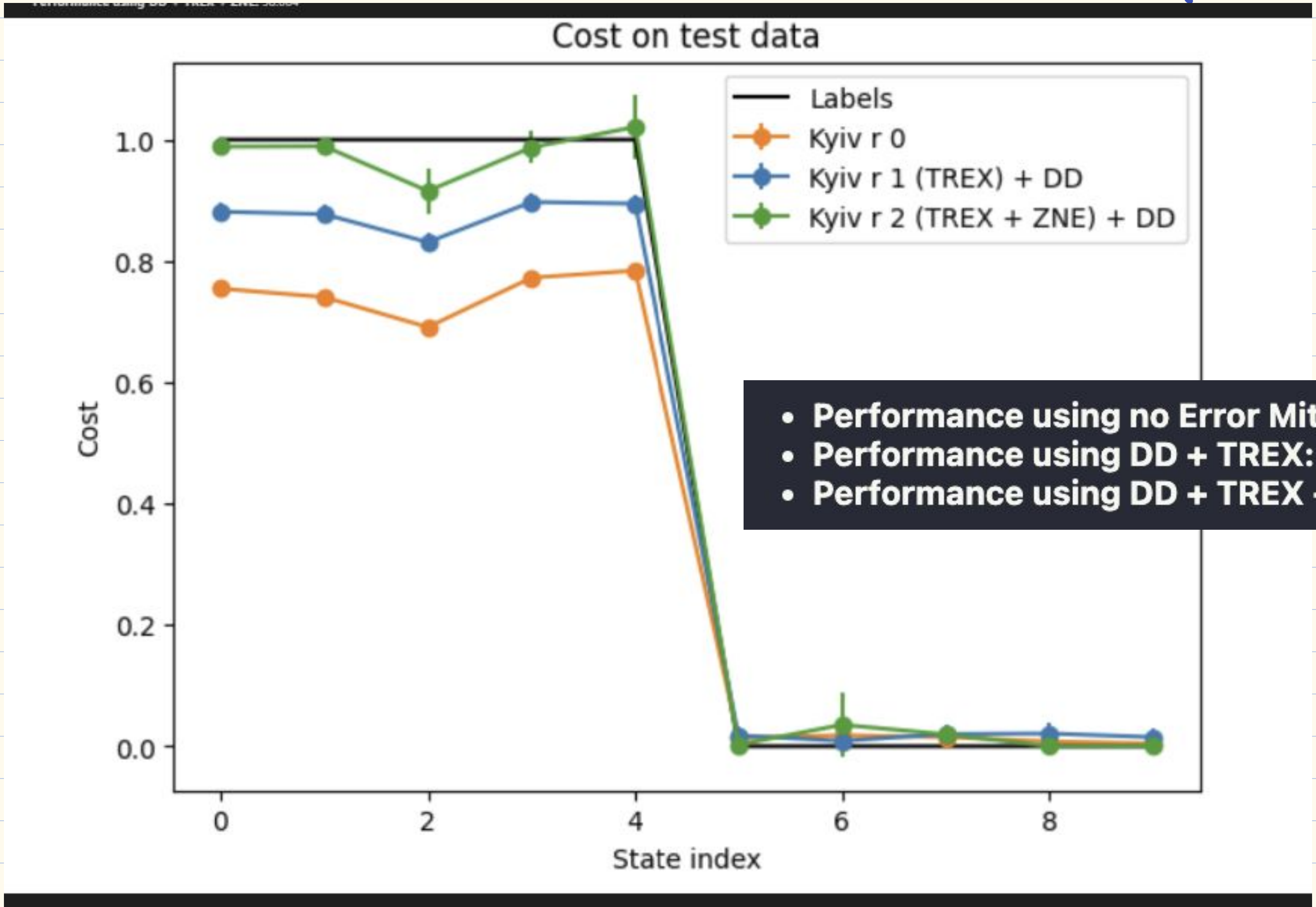
Testing Error Mitigation Techniques

```python
## No DD, no TREX (no ZNE)
options_0 = EstimatorOptions(
    default_shots= 5000,
    optimization_level= 0,
    resilience_level= 0,
    dynamical_decoupling = {'enable': False, 'sequence_type': 'XpXm'},
    twirling= {'enable_measure': False}
)#Add your code here

## DD + TREX (no ZNE)
options_1 = options_1 = EstimatorOptions()
options_1.optimization_level =0
options_1.resilience_level = 1
options_1.default_shots = 5000
options_1.dynamical_decoupling.enable = True
options_1.dynamical_decoupling.sequence_type = 'XpXm'
options_1.twirling.enable_measure = False #Add your code here
✓ 0.0s
```

Testing Error Mitigation Techniques

```
estimator = Estimator(backend=backend, options=options)
pm = generate_preset_pass_manager(optimization_level=3, backend=backend)

pubs = []
for bird, label in enumerate(list_labels):
    ### Write your code below here ###


    qc = amplitude_embedding(5, index)  # Assuming 5 qubits

    classifier = qc.compose(anstaz)
    transpiled_classifier = pm.run(classifier)
    transpiled_obs = pm.run(obs)
    ### Don't change any code past this line ###
    pub = (transpiled_classifier, transpiled_obs, opt_params)
    pubs.append(pub)

job = estimator.run(pubs)
job_id = job.job_id()
print(f"Job ID: {job_id}")
print(f"Status: {job.status()}")

return job_id
```

Testing Error Mitigation Techniques, Results



- **Performance using no Error Mitigation:** 86.824
- **Performance using DD + TREX:** 92.979
- **Performance using DD + TREX + ZNE:** 98.004

Thank You